THE PARALLEL UNIVERSE

Issue 4
Fall 2010

MASTER MULTICORE

Introducing Intel® Parallel Studio 2011
by Leila Chucri

Intel® Cilk Plus for Fine-Grained Parallelism
by Krishna Ramkumar

The World's First Sudoku 'Thirty-Niner'
by Stephen Blair-Chappell

CONTENTS

Letter from the Editor

Breaking Ground and Building Trust, BY JAMES REINDERS 4

James Reinders, lead evangelist and director of Intel® Software Development Products, addresses recent innovations in apps and tools, highlights key 2010 milestones, and explores what's next in the new year and beyond.

Intel® Parallel Studio 2011: Getting Your App Ready for Multicore Just Got Easier BY LEILA CHUCRI 6

Engineer Leila Chucri introduces Intel® Parallel Studio 2011, examining new features with an eye to enabling the development environment for multicore.

Using Serial Modeling Tools to Tame the Parallel Beast, BY JOHN PIEPER 12

Addressing many of the key reasons why parallelism is considered difficult, Intel® Parallel Advisor offers a proven threading methodology, and enables parallel and serial modeling.

Intel® Cilk Plus, BY KRISHNA RAMKUMAR 16

Intel® Cilk Plus adds fine-grained task parallelism support to C and C++, making it easy to add parallelism to both new and existing software, and efficiently exploit multiple processors.

Nine Tips to Parallel Programming Heaven

BY STEPHEN BLAIR-CHAPPELL 19

In this interview, Dr. Yann Golanski shares with us his favorite tips on parallel programming. The tips are based on investigative work on parallel n-body simulation code carried out during his doctoral studies.

The World's First Sudoku* "Thirty Niner"

BY STEPHEN BLAIR-CHAPPELL 20

Lars Peters Endresen and Håvard Graff, two talented engineers from Oslo, share with us how they created what may be the world's first Sudoku* puzzle that has 39 clues.

BREAKING GROUND AND BUILDING TRUST

LETTER FROM THE EDITOR

James Reinders is Chief Software Evangelist and Director of Software Development Products at Intel Corporation. His articles and books on parallelism include *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism.*

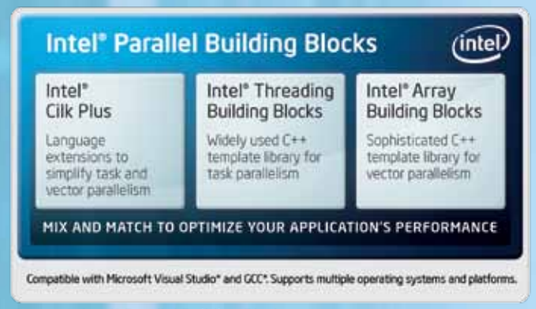


Figure 1. Intel® Parallel Building Blocks

Multicore processors have brought parallel computing into the mainstream. Software developers are embracing parallelism tools and programming models.

In September 2010, after a successful summer of beta testing with customers around the world, we are introducing Intel® Parallel Studio 2011 (see the article on Intel Parallel Studio 2011 in this issue), which includes a range of new features, including:

- A comprehensive set of parallelism development models – Intel® Parallel Building Blocks (Intel® PBB)
- An innovative threading assistant – Intel® Parallel Advisor 2011
- Full support for Microsoft Visual Studio 2010*, as well as 2005 and 2008
- Premier technical support

Both Intel Parallel Studio 2011 and Intel PBB build on prior products, provide even more support for parallel programming, and are backward compatible.

When we decide to rely on a product, we are looking for signs that it is worthy of our trust. The key for me is figuring out the commitment level of the company behind the product, and of those who are already using it.

I have a great job because I'm working with products that have a high commitment level from Intel and very committed customers.

Our most popular parallelism project is Intel® Threading Building Blocks (Intel® TBB), which has been adopted widely. One of the most recent expressions of trust for Intel TBB was its inclusion in Adobe Creative Suite 5*. The reliable portability and outstanding performance of Intel TBB has proven valuable to software developers on many operating systems and processors.

The groundbreaking Intel® Parallel Studio is now demonstrating value by helping software developers on Windows* tap into the power of multicore processors.

It is my distinct pleasure to see us extend our product base and capabilities in a natural way to serve both current and new customers.

Intel PBB augments the broad foundation of Intel TBB with two

Design Phase	Bug and Debug Phase	Verify Phase	Tune Phase
INNOVATIVE THREADING ASSISTANT	**COMPILER AND THREADED LIBRARY**	**MEMORY AND THREADING ERROR CHECKER**	**THREADING AND PERFORMANCE PROFILER**
Intel® Parallel Advisor	**Intel® Parallel Composer**	**Intel® Parallel Inspector**	**Intel® Parallel Amplifier**
Threading design guide tool simplified, and speeds parallel application design › Identifies the areas in applications that can benefit the most from parallelism › Provides step-by-step guidance for threading applications	Optimizing compiler boosts performance and threaded application design › C++ Compiler and libraries › Code Coverage › Debugger › Intel® Parallel Building Blocks—Set of comprehensive parallel models that supports multiple ways to exploit parallelism	Error detection analysis tool for higher code reliability and quality › Error detection analysis tool for higher code reliability and quality › Finds memory leaks and corruption › Finds data races and deadlocks	Tuning analysis for optimized and scalability › Tuning analysis for optimized and scalability › Performance and scalability analysis › Locks and waits analysis

Fig.2: Intel Parallel Studio 2011

significant additions: the compiler-assisted capabilities of Intel® Cilk Plus and the sophisticated vector parallelism capabilities of Intel® Array Building Blocks (Intel® ArBB) (see the article on Intel Cilk Plus in this issue). Intel TBB continues to supply key functionality, including a thread-aware memory allocator, concurrent containers, portable locks and atom operations, a task-stealing scheduler, and a global timer.

Intel TBB has not stood still; the latest version includes support for FIFO scheduling, additional support for C++ 0x, a new container (concurrent_unordered_map), a "Design Patterns" manual, and Microsoft Visual Studio 2010* support. Also available is a community preview feature called tbb::graph, which is a unified approach that lets applications use graphs to express parallelism. This is a versatile feature that can be used to implement anything from simple static dependency graphs to complex message-passing, actor-like graphs. It is an exciting new feature that adds substantial coordination capabilities.

Intel Cilk Plus is made up of four things:

- Three simple but very powerful keywords to express parallelism: cilk_for, for parallel loops, and cilk_spawn + cilk_sync for parallel function invocation
- A solution, called hyperobjects, to deal with contention for shared variables by replicating them as local copies to match tasks created by the new keywords, and providing for their reduction back to a shared value at the conclusion of a parallel computation
- Array notations for data parallel operations on array slices, such as a[] = b[] + c[] to sum two vectors, or a[][1] = sqrt(b[][2])
- Array notations used with elemental functions to allow computations such as __sec_map(saxpy, 2.0, x[0:n], y[0:n])

With Intel Cilk Plus, we have focused a compiler implementation. Unlike Intel TBB, Intel Cilk Plus involves the compiler in optimizing and managing parallelism. Benefits include: (1) code that is easier to write and comprehend because it is better integrated into the language through use of keywords and native syntax, and (2) compiler optimizations that allow for more avoidance of data race conditions, and slightly better performance.

The compiler understands these four parts of Intel Cilk Plus, and is therefore able to help with compile-time diagnostics, optimizations, and runtime error checking. Intel Cilk Plus has an open specification, so other compilers may also implement these exciting new C/C++ language features.

With Intel ArBB, we have focused on offering very sophisticated vector parallelism capabilities in an easy-to-program and highly portable fashion. Features include the ability to use parallelism from SIMD instructions, multicore, and manycore processors. Intel ArBB offers built-in support for regular, irregular, and sparse matrices. These features allow a program written in mathematical terms to harness parallelism for performance without undue effort by the developer.

Intel Parallel Studio 2011 builds on the original Intel Parallel Studio (released in May 2009) by augmenting it with Intel® Parallel Advisor and Intel PBB (see the article on Intel Parallel Advisor 2011 in this issue). Intel Parallel Advisor helps software architects consider different possible approaches to adding parallelism to an existing application without having to fully implement the parallelism. This can greatly reduce the time spent evaluating implementation options and choosing the right one for your application.

James Reinders
September 2010

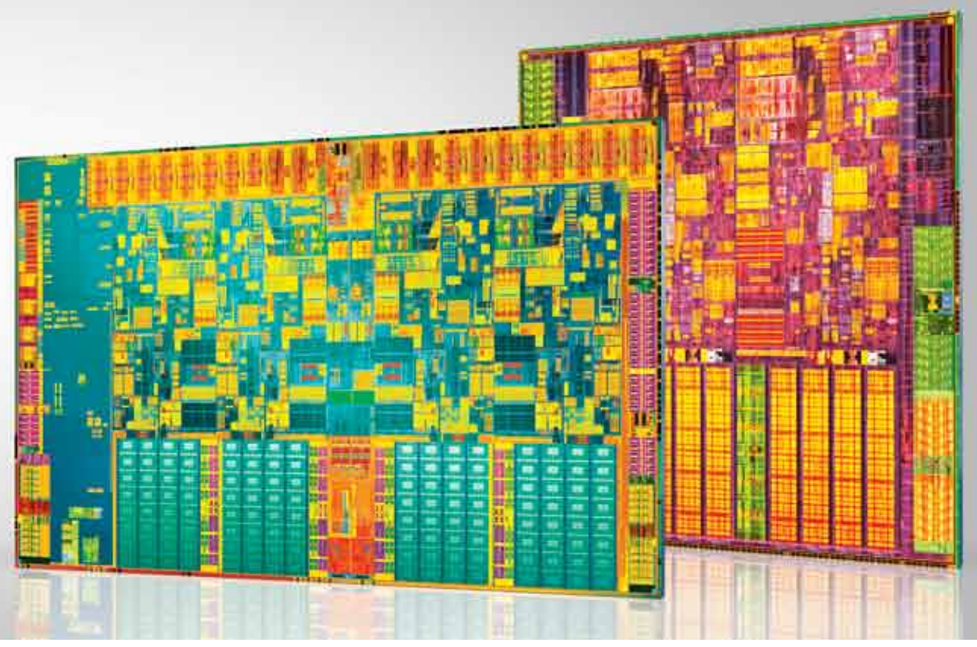
THE PARALLEL UNIVERSE

Getting Your App Ready for Multicore Just Got Easier

Intel® Parallel Studio 2011

By Leila Chucri
Product Marketing,
Intel Corporation

Leila Chucri introduces Intel® Parallel Studio 2011, examining new features with an eye to enabling the development environment for multicore.



Multicore: The Game Changer

Once relegated to a small niche of performance-hungry apps like gaming, parallel programming is making speedy inroads with mainstream developers. Designing applications that exploit the performance and scalability attributes of multicore technology is no longer considered "nice to have" functionality. Today, every developer knows that as the number of processor cores increases, serial applications can no longer rely on the latest processor chip to automatically provide greater headroom or a better user experience. Parallel programming enables applications to fully utilize multicore innovations and stay competitive. Whether optimizing serial code or creating new applications for multicore, parallel programming has evolved to improve proficiency with flexible design models and tools that make applications reliable and fast.

Making Parallel Application Design – Simple, Efficient, and Fast

Over a year ago, Intel launched the first product to ease mainstream developers adopting multicore. Intel® Parallel Studio was the first development tool to enable auto-scaling features through a comprehensive set of compilers, threaded libraries, memory error detection, and analysis tools. Although other companies have jumped on this new development environment, including Microsoft with Visual Studio PPL*, **Intel maintains its strong leadership with the introduction of Intel® Parallel Studio 2011.** The latest edition includes the first tool to guide a developer on where their application can benefit most from multicore, plus a new set of task and data parallel models. This quick overview introduces some of the noteworthy new features of Intel Parallel Studio 2011.

Intel Parallel Studio 2011 Revealed

The advanced tools and methodologies of Intel Parallel Studio 2011 help developers thrive in the multicore world. It enables Microsoft Visual Studio 2005*, 2008*, 2010* C/C++ developers to easily ramp up to parallelism and achieve the application performance and scalability benefits of multicore processor technology today, and in the future.

What's New

- **Expanded Threading Options with Intel® Parallel Building Blocks (Intel® PBB):** Intel® Parallel Building Blocks is a comprehensive and complementary set of parallel models that provide solutions ranging from general-purpose to specialized parallelism. It enables developers to mix and match parallel models within an application to suit their specific environment and needs, providing a simple yet scalable way to develop for multicore.
  - **Intel® Threading Building Blocks 3.0:** The widely adopted, award-winning C++ template library solution provides flexible, cross-platform parallelism solution enabling general parallelism.
  - **Intel® Cilk Plus:** Built into the Intel® C/C++ Compiler, it provides an easy-to-use, well-structured model that simplifies parallel development, verification, and analysis.
  - **Intel® Array Building Blocks:** An API backed by a sophisticated runtime library, it provides a generalized data parallel programming solution that frees developers from dependencies on particular low-level parallelism mechanisms or hardware architectures. (Available in beta at: software.intel.com/en-us/data-parallel/)
- **Intel® Parallel Advisor:** An innovative threading guide simplifies and demystifies parallel application design
- **Microsoft Visual Studio 2010 support**

"Intel® TBB was surprisingly quick and simple to implement, and made the Simul Weather SDK really fly on the Intel® Core™ i7 processor. With close to linear scaling, Simul Weather* and Intel's tools open up great opportunities for game developers to integrate dynamic weather and clouds."

Roderick Kennedy,
Principal and founder of Simul Software

Intel® Parallel Advisor: An innovative threading guide simplifies and demystifies parallel application design.

Take the easy path to parallelism with Intel Parallel Advisor to design your parallel application and tap into the performance and scalability advantages of multicore.

Whether you are new to parallelism or currently developing parallel applications, you will find Intel® Parallel Advisor an essential tool to use during the design and discovery phase of parallelizing code.

Intel Parallel Advisor offers visibility to developers on where to consider parallelism in C/C++ code; helps developers identify and experiment with parallel opportunities; exposes performance benefits; identifies how to restructure the application, as well as what time saving tools to use to debug and tune the application; and enables developers to evaluate the return on their investment before committing to the process of parallelizing their application.

Intel Parallel Advisor gives developers a clear roadmap for parallelizing applications, with step-by-step guidance for threading applications using widely adopted, proven abstractions for parallelism; provides the steps for task-oriented parallelism; and targets coarse-grain parallelism.

Design Phase

Intel® Parallel Advisor analyzes the executing serial program as developers progress through the methodology.

- Survey Target—Helps focus on hot call trees and loops as locations to experiment with parallelism.
- Annotate Sources—Enables developers to insert Parallel Advisor annotations into their sources to describe parallel experiments.
- Check Suitability—Helps evaluate the performance of their parallel experiment by displaying the performance projection for each parallel site and how each site's performance impacts the entire program.
- Check Correctness—Assists developers by identifying data issues such as races in the parallel experiment.

"The Intel® Parallel Advisor design approach was instrumental in introducing parallelism into our code. The survey feature helped improve our code by finding areas in our serial code that took a lot of CPU time, and where our code would benefit from parallelism."

Dr. William Orttung,
Emeritus Professor of Chemistry

"Intel® Parallel Advisor reduces the risk of adding parallelism, since it helps focus effort in the right place, avoiding wasted implementation effort."

Matt Osterberg,
Vickery Research Alliance

"As a Microsoft Visual Studio* C++ developer for many years, and with no previous experience with parallel programming, Intel® Parallel Advisor proved to be a major boon in making it easier and efficient to implement parallelism through the planning and production phase."

Brian Reynolds,
Brian Reynolds Research

Intel® Parallel Composer 2011 C/C++ Compiler boosts performance with libraries, parallel models, and debugging capabilities.

Intel® Parallel Composer 2011 – C/C++ optimized compiler with Intel® Integrated Performance Primitives' advanced libraries and Intel® Parallel Building Blocks (Intel® PBB), a set of comprehensive parallel development models. Together, these tools help enhance performance and streamline parallel application development.

Intel Parallel Composer helps simplify adding parallelism with the newest versions of Intel® Compilers and libraries for parallelism and thousands of lines of ready-to-use code. Supports the entire spectrum of parallel expression from simple to complex, data to task, by offering application-class-specific, pre-threaded, and thread-safe libraries. Saves time and takes advantage of multicore processors with automatic functions.

Build and Debug Phase

Optimizing Compiler with Libraries Boosts Application Performance

"Here at Trading Systems Lab, we got a 10% to 20% performance boost in the multimode trading simulator that's used in our TSL Algo Auto-Design Platform by using the C++ compiler in Intel® Parallel Studio. The compatibility with Microsoft Visual C++* is great, and we're looking forward to using more parallelism features in Parallel Studio."

Mike Barna,
President Trading Systems Lab

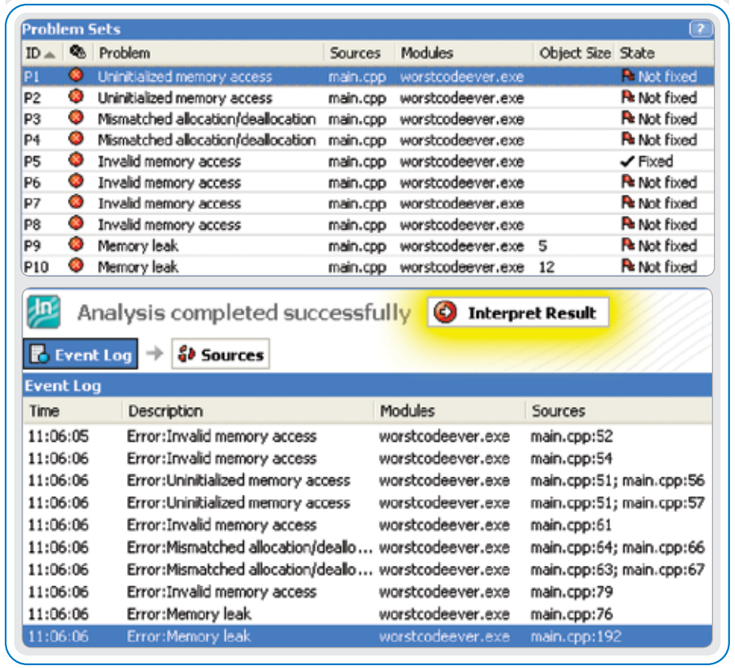
As you evolve your code to parallelism through the design phase, you also need future-proof tools to further optimize, verify, and tune your application to get the full benefits of the performance and scalability attributes of multicore and manycore technology. Intel Parallel Studio 2011 provides developers with critical assistance throughout the development lifecycle, from design, code and debug, and verification, to tuning application performance and reliability.

Intel® Parallel Inspector 2011 with memory and threading error checker enhances application reliability.

Catching software defects early in the development cycle can save you time and cost, and increase your ROI. Intel® Parallel Inspector provides a comprehensive solution for serial and multithreading error checking. Pinpoints memory leaks and memory corruption as well as thread data races and deadlocks.

Verify Phase

Dynamic Analysis Memory and Threading Checker



"The performance benefits of multicore and manycore are critical to SIMULIA's business. Intel® Parallel Inspector provides a powerful way to develop parallel code compared to traditional methods, which can be lengthy and costly– especially if the price of unstable code is paid by the customer."

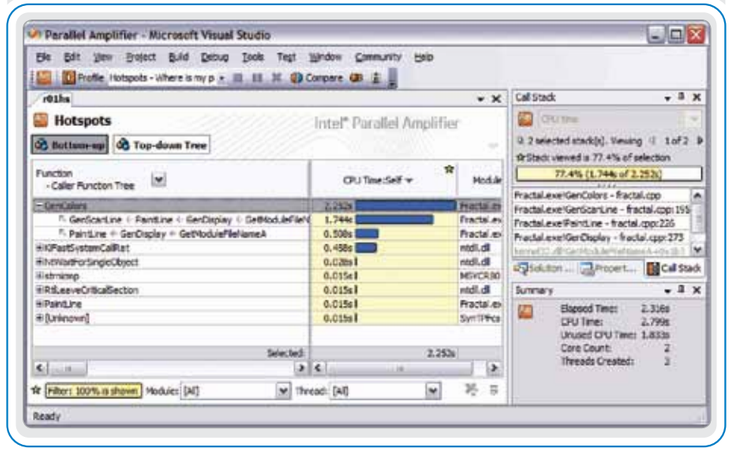
Matt Dunbar,
Chief Architect SIMULIA

The Intel® Parallel Amplifier 2011 threading and performance profiler enhances performance.

By removing the guesswork and analyzing performance behavior in Windows* applications, Intel® Parallel Amplifier 2011 provides developers quick access to scaling information for faster and improved decision making. Fine-tune for optimal performance, ensuring cores are fully exploited and new capabilities supported.

Tune Phase

Threading and Performance Profiler



"Thanks Intel, you guys rock! I decided to give Intel® Parallel Amplifier a run. I was delighted when it pointed me to the right source line that was taking much of the time. I made the change, and voilà, our app is now almost 10 times faster. The GUI is very easy to use, in my opinion."

Dat Chu,
Research Assistant Computational Biomedicine Lab
University of Houston

USING SERIAL MODELING TOOLS TO

TAME THE PARALLEL BEAST

By John Pieper
Senior Staff
Software Engineer

Addressing many of the key reasons why parallelism is considered difficult, Intel® Parallel Advisor offers a proven threading methodology, and enables parallel and serial modeling.

A lot of programmers seem to think parallelism is hard.

A quick Internet search will yield several blogs talking about the difficulty of writing parallel programs (or parallelizing existing serial code). There do seem to be many stumbling blocks waiting to trip up the novice. Here's a representative list (based on one by Anwar Ghuloum):

- **Finding the parallelism:** This can be difficult because in tuning code for serial performance, we often use memory in ways that limit the available parallelism. Simple fixes for serial performance often complicate the original algorithm and hide the parallelism that is present.
- **Avoiding the bugs** (data races, deadlocks, and other synchronization problems): Certainly there is a class of bugs that affect parallel programs that serial programs don't have. And in some sense, they are worse, because timing-sensitive bugs are often hard to reproduce—especially in a debugger.
- **Tuning performance** (granularity, throughput, cache size, memory bandwidth): Serial programmers have to worry about memory locality, but for parallel programs, the programmer also has to consider the parallel overheads, and unique problems like false sharing of cache lines.
- **Future proofing:** Serial programmers don't worry about whether the code they are writing will run well on next year's processors—it's the job of the processor companies to maintain upward compatibility. But parallel programmers need to think about how their code will run on a wide range of machines, whether there are one or two, or many processors. Software that is tuned for today's quad-core processors may still be running unchanged on future 16-, 32-, or even 64-core machines.
- **Using modern programming methods:** Object-oriented programming makes it much less obvious where the program is spending its time.

"When the program has fully evolved, the result is a correct serial program with annotations describing a parallelization with known good performance and no synchronization issues."

Other reasons that parallel programming is considered hard include the complexity of the effort, not enough help for developers unfamiliar with the techniques, and a lack of toolsfor dealing with parallel code. When adding parallelism to existing code, it can also bedifficult to make all the changes needed to add parallelism all at once, and to ensure that there is enough testing to eliminate timing-sensitive bugs.

Intel® Parallel Advisor is a new tool for developers who need to add parallelism to existing serial code, and it helps to address these problems. Intel Parallel Advisor makes it easier to parallelize. It increases the programmer's return on investment by focusing effort where it matters. It helps the programmer identify problems early, so that little effort is wasted on unproductive directions. The key to Intel Parallel Advisor's success is its reliance on a well-proven method of introducing parallelism: serial modeling.

"By modeling parallel overheads for a range of processors, Intel® Parallel Advisor encourages the programmer to resolve scaling issues at the design stage."

Evolving serial code to parallel with Intel Parallel Advisor

Intel Parallel Advisor leads the programmer with a proven threading method. This method was developed by Intel engineers over years of working with customers to thread their applications. The key to the method is to consistently check and resolve problems early, while slowly evolving the code from pure serial, to serial but capable of being run in parallel, to parallel.

The first step is to measure where the application spends time—effort spent in hot areas will be effective, while effort spent elsewhere is wasted. The next step is to insert annotations to describe a potential parallelization. The annotations form an experiment: **what would happen if this code ran in parallel?** The annotations do not create actual parallelism, but describe where in the serial program parallelism could be inserted. Intel Parallel Advisor observes the execution of the program with annotations, and uses the serial behavior to predict the performance and bugs that might occur if the program actually executed in parallel.

Intel Parallel Advisor models the performance of the theoretical parallelism. This allows the programmer to check the effect of overheads of parallelization. Checking early in the process, while the program is still serial, ensures that time is not wasted on parallelizations that are doomed to poor performance. Either the annotations are changed to model a different parallelization that resolves the performance issue or, if no alternative is practical, the programmer can focus effort on a more profitable location.

Intel Parallel Advisor also models the correctness of the theoretical parallel program described by the annotations. This lets the tool detect race conditions and other synchronization errors, while still running the serial program. Because the program still runs serially, it is easy to debug and test, and computes the same results. The programmer can change the program—either the annotations or the code itself—to resolve the potential races. After each change, the program remains a serial program (with annotations) and can be tested and debugged using the normal processes.

When the program has fully evolved, the result is a correct serial program with annotations describing a parallelization with known good performance and no synchronization issues. The final step in the process is to convert those annotations to parallel code. After conversion, the parallel program can undergo final tuning and debugging with the other tools in Intel® Parallel Studio.

The value of the Intel Parallel Advisor method

Intel Parallel Advisor addresses many of the key reasons why parallelism is considered difficult. Evolving a program step- by-step is much easier than attempting to convert all at once. Taking small steps and testing after each change allows a more comfortable development model. It also reduces risk—at any point, the parallelization effort can be suspended and resumed later. The partially evolved code is still a correct serial program. In fact, it's often a simpler, easier-to-understand program, since the changes to enable parallelism tend to regularize and simplify the code.

Focusing on the hot parts of the program ensures that time is spent effectively. It also encourages the programmer to consider coarse-grained task parallelism, which is more likely to result in good scaling. Intel Parallel Advisor's performance model builds on this approach. By modeling parallel overheads for a range of processors, Intel Parallel Advisor encourages the programmer to resolve scaling issues at the design stage.

Serial modeling gives the programmer the best of both worlds. The code remains serial, computes the same answer, and uses the same test system, even as it evolves to parallelism. The serial modeling tools allow the programmer to fix problems that might occur in the parallel program without the difficulty inherent in non-deterministic execution.

When the method "fails"

Another advantage of the Intel Parallel Advisor method becomes evident when we consider what happens if the parallelization effort fails. If the programmer is unable to find a correct, high-performing parallelization, the annotations can be deleted or commented out. Further, since the effort of adding annotations and modeling performance and correctness is much lower than the effort of implementing parallelism, the end result—the original serial program—is reached much more quickly with Intel Parallel Advisor, so the amount of "wasted" effort is minimized. □

Design it right the first time with Intel® Parallel Advisor.

Intel Parallel Advisor is available as part of Intel® Parallel Studio. Download an evaluation kit, and see how easy parallelism can be.

THE PARALLEL UNIVERSE

INTEL® CILK PLUS

By Krishna Ramkumar
Technical Consulting Engineer,
Intel Corporation

Intel® Cilk Plus adds fine-grained task parallelism support to C and C++, making it easy to add parallelism to both new and existing software, and efficiently exploit multiple processors.

Introduction

It has been five years since we had free lunches. I am referring to the legendary article by Microsoft's Herb Sutter on the multicore revolution and the imperative to write software that takes advantage of multiple execution units on the same chip or die. As Sutter points out, this entails a fundamental turn or a paradigm shift in programming for the multicore era. From a software development perspective, it clearly means that developers have to revisit the implementation of their software and "parallelize" their erstwhile serial application. They have to decompose their application into tasks, and identify tasks that are independent of each other so that such tasks can be run in "parallel" (you may read it as "at the same time"). Also, with hardware support for vectorization constantly improving, it is definitely wise to harness data parallelism in your program. In order to take advantage of multicore platforms with advanced vectorization capabilities, easy and efficient programming models are required to realize both task and data parallelism, which could potentially result in faster execution of code.

Intel's software group provides a range of tools specifically designed to help developers parallelize their applications. Intel® Parallel Building Blocks (Intel® PBB) is a comprehensive and complementary set of models that provide solutions ranging from general purpose to specialized parallelism, and includes Intel® Cilk Plus, Intel® Threading Building Blocks (Intel® TBB) and Intel® Array Building Blocks. This article focuses on the concepts and features of Intel Cilk Plus.

Intel® Cilk Plus

Intel Cilk Plus is a programming methodology/platform designed to enable developers to take advantage of today's multicore computer systems. It is the result of nearly two decades of research by some of the industry's smartest minds. Intel Cilk Plus is a technology that adds task and data parallelism support to C and C++, making it easy to add parallelism to both new and existing software to efficiently tap the potential of multiple processors.

Intel Cilk Plus is particularly well suited for, but not limited to "divide and conquer" algorithms. This strategy solves problems by breaking them into sub-problems (tasks) that can be solved independently and then combining the results. Recursive functions are often used for divide-and-conquer algorithms, and are well supported by Intel Cilk Plus. Tasks can either be functions or iterations of a loop. Intel Cilk Plus keywords identify function calls and loops that can run in parallel. The Intel Cilk Plus runtime schedules these tasks to run efficiently on the available processors.

From a C/C++ developer perspective, it is a simple yet powerful model because it adds very little overhead in transforming an existing serial program to a parallel implementation. This simplicity may seem limiting at first when compared to OpenMP* or Intel TBB, both of which come with many parallel programming constructs. However, Intel Cilk Plus has all the features needed to support the implementation of parallel algorithms. Its sophistication is in its minimalism.

Intel Cilk Plus requires minimal to no intrusion into the source code, making a serial program just a few keywords away from parallelism, thus easing the transition to parallelism for programmers trained to solve problems serially. In fact, a program parallelized (and which is race free) using Intel Cilk Plus technology can be interpreted as a serial program if the keywords are ignored. This makes such a program easy to debug and test. Please refer to the code samples in the final section for an illustration on the use of Intel Cilk Plus keywords.

Intel Cilk Plus' semantics allow you to design a program without specifying the number of processors on which the program will execute. The runtime scheduler takes care of efficiently load balancing your program across however many processors are available. When a portion of your parallel computation executes on a single processor, Intel Cilk Plus can execute it just like ordinary C code, taking full advantage of all the compiler optimizations and runtime efficiencies that a good C system offers. By starting from good single-core performance, Intel Cilk Plus ensures that a program with sufficient parallelism gets good speedup (an increase in performance) whether it is run on a large number of processors or just a few.

In practice, most Intel Cilk Plus programs require few (if any) locks. Its parallel control constructs obviate the need for the locks that many other parallel programming models require for interthread communication and synchronization. In many common situations where locking would seem to be necessary to avoid a race, Intel Cilk Plus provides hyperobjects, novel data structures that resolve races on global variables without sacrificing performance or determinism.

By avoiding locks and using hyperobjects, Intel Cilk Plus sidesteps many performance anomalies caused by locks, such as lock contention, which can slow down parallel programs significantly; and deadlock, which may cause your application to freeze. In fact, Intel Cilk Plus doesn't even have mutex locks of its own. You can use Intel TBB locks if you ever need one. However, locking works against parallelism. In other words, if your application suffers from lock contention, then it means that you have to revisit your algorithm and see if it is possible to restructure your data so as to make it amenable to parallelism.

By using reducers, you only need to identify the global variables as reducers where they are declared. No logic needs to be restructured. In contrast, most other concurrency platforms have a hard time expressing race-free parallelization of this kind of code. The reason is that reductions in most languages are tied to a control construct. For example, reduction in OpenMP is tied to the parallel for loop pragma. Moreover, the set of reductions in OpenMP is hardwired into the language, and in particular, list appending is not supported.

Relationship/Co-existence with Intel TBB

Intel TBB is a widely used C++ template library for parallelism that extends C++ by abstracting away thread management, thereby allowing straightforward parallel programming. A full-blown discussion of Intel TBB is beyond the scope of this article. The interested reader could pick up a copy of the book titled *Intel Threading Building Blocks* written by Intel's James Reinders and published by O'Reilly, which has a comprehensive treatment of the concepts and features of Intel TBB.

The underlying symmetry in the runtime shared by Intel Cilk Plus and Intel TBB is extremely useful for developers. While Intel Cilk Plus is suitable for both procedural as well as an object-oriented codebase, Intel TBB can also be used to parallelize C++ code where portability to non-Intel compilers is required. Intel Cilk Plus and Intel TBB are different interfaces for similar underlying threading runtime implementation. Both implement work/task stealing where the runtime steals work from chunks that have busy workers and runs them on chunks that have idle workers. This is what makes software developed using Intel Cilk Plus and Intel TBB scalable.

When it comes to programming in parallel, the importance of synchronization primitives, thread-safe container data structures, and scalable memory allocators cannot be overemphasized. Thanks to Intel TBB being implemented as a C++ standards-compliant portable library, a program written using Intel Cilk Plus can use all of the above features, which are implemented by Intel TBB in an elegant fashion. We already discussed above that Intel Cilk Plus can use Intel TBB's synchronization primitives if needed. Intel Cilk Plus can also use Intel TBB's thread-safe containers and scalable memory allocators.

Exploiting data-parallel hardware

More often than not, in software programs that massage a lot of data there is scope for processing the data in parallel when the same operation is to be performed on all of the data. Fortunately, modern computer systems come with additional instruction sets and associated registers that are capable of processing one operation on multiple pairs of operands at the same time. This is called vectorization. The Intel® Compiler automatically analyzes a program and generates vector code that exploits vector hardware capabilities. Besides automatic vectorization, it also has features that let you explicitly enforce vectorization in the form of compiler directives and C/C++ language extensions. These features can be used along with any programming platform such as Intel Cilk Plus, Intel TBB, or OpenMP, which gives developers a lot more choice and flexibility.

The #pragma simd compiler directive can be used for loop vectorization. The simd pragma is used to guide the compiler to vectorize a loop. Vectorization using the simd pragma complements (but does not replace) the fully automatic approach.

The C/C++ language extensions (part of Intel Cilk Plus) provide explicit data parallel array notations so that the compiler can perform transformations to generate vectorized code. As an example, the following array notation can be used if element-wise multiplication of arrays b and c is to be performed.

```
a[:] = b[:] * c[:];
```

To summarize, Intel Cilk Plus is a language extension with support for task parallelism so as to make full use of multiple cores and data parallelism that can use vectorization capabilities within each core. Combined, these extensions can harness the power of modern microprocessors.

Intel Cilk Plus is part of Intel® Parallel Composer. We encourage you to try Intel Cilk Plus by downloading Intel® Parallel Composer from the following link: http://software.intel.com/en-us/articles/intel-software-evaluation-center/. The download also includes documentation that provides detailed reference information about Intel Cilk Plus. □

Examples illustrating use of Intel Cilk Plus keywords

Listing 1: cilk_spawn and cilk_sync

```
void mergesort(int a[], int left, int right)
   {
       if(left < right)
       {
           int mid = (left + right)/2 ;

           /* The below calls to the mergesort function are independent of each
           other. In other words, they are task parallel. This parallelism can
           easily be expressed by using the keyword cilk_spawn as shown below. */

           cilk_spawn mergesort(a, left, mid);
           mergesort(a, mid+1, right);

           /* Use of cilk_sync ensures that the threads spawned in this
           function join at this point. Serial execution resumes after this point */

           cilk_sync;
           merge(a, left, mid, right);

       }

   }
```

Listing 2: cilk_for

```
/* The below cilk_for keyword enables parallel loop iterations. If the
cilk_for keyword is replaced by for, it is then an ordinary for loop */

cilk_for (int i = begin; i < end; i += 2)
{
       foo(i);
}
```


NINE TIPS TO PARALLEL PROGRAMMING HEAVEN

By Stephen Blair-Chappell
Intel Compiler Labs

In this interview, Dr. Yann Golanski shares with us his favorite tips on parallel programming. The tips are based on investigative work on parallel n-body simulation code carried out during his doctoral^ studies.

A cloud of cold interstellar gas

The Formation of Stars

It is thought that stars are formed from the Inter Stellar Medium (ISM), an area populated with particles of predominantly hydrogen and helium. Within the ISM, there are dense clouds. These clouds are normally in equilibrium, but can collapse if triggered by various events.

In the research work done by Dr. Golanski, the model simulates the collapse of the ISM, by seeding the ISM with coolant from a supernova.

The collapsing cloud continues collapsing until equilibrium is reached. This cloud is known as a protostellar cloud.

Further contraction and fusion of the protostellar cloud takes place, resulting in the eventual formation of a star.

1 Just buy a faster machine.

First look at how much it will cost you to make your program parallel. If it will take, say two months of coding, can you just buy a faster machine that will give you the speedup you want? Of course, once you reach the limits of a machine's speed, you are going to have to do some parallelization.

2 Start small.

Don't try to make everything parallel at once, just work on small bits of code.

3 Starting from scratch? Use someone else's wheel.

If you are starting from scratch, see what other people have done. Learn from others. Don't reinvent the wheel.

4 Find a way of logging\ debugging your application.

Make sure you have a way of tracing what your application is doing. If necessary, buy some software tools that will do the trick. Prints on their own will probably not help.

5 Look at where the code is struggling.

Examine the runtime behavior of your application. Profile the code with Intel®\VTune™ Performance Analyzer. The hotspots you find should be the ones to make parallel.

6 Write a parallel version of the algorithm.

Try rewriting the algorithm so it is parallel-friendly.

7 Stop when it's good enough.

When you think it's good enough, stop. Step back, go for a pint. Have set goals—when you've achieved them, you're done.

8 Tread carefully. You are walking on eggs, and some eggs are land mines.

Take care with the parallel code. Some innocent errors could blow up your program. Use a good tool to check for any data races and other parallel errors.

9 Get the load-balancing right.

Once you've made your code parallel, make sure all the threads are doing equal amounts of work

The tips were recorded over a pleasant Thai meal in the City of York. Between the various dishes, Dr. Golanski spoke about the advice he'd give to someone starting to parallelize an application. At the end of the meal, the restaurant owner asked if we would mention the restaurant. Well here goes—if you are ever in the center of York, look for Siam House on Goodramgate.

^Y. Golanski and M. M.Woolfson. A smoothed particle hydrodynamics simulation of the collapse of the interstellar medium. Monthly Notices Royal Astronomical Society. 320, 1-11 (2001).

The complete case study, along with hands-on examples, will be available in the WROX book, *Parallel Programming with Intel® Parallel Studio.* Stephen Blair-Chappell and Andrew Stokes, Wiley Publishing Inc. ISBN 9780470891650 (March, 2011)

Photo credit: NASA, ESA, and M. Livio and the Hubble 20th Anniversary Team (STScI)

The World's First Sudoku* 'Thirty-Niner'

By Stephen Blair-Chappell
Intel Compiler Labs

Lars Peters Endresen and Håvard Graff, two talented engineers from Oslo, share with us how they created what may be the world's first Sudoku puzzle that has 39 clues.

The Nature of the Challenge

There are over 6 x 10^{21} valid Sudoku boards in a 9x9 grid. Using brute force to try all the combinations of numbers is not that difficult a programming exercise, the real challenge is in getting a program that will complete the calculations within the lifetime of the programmer!

The challenge our engineers squared up to, was how to produce a puzzle with 39 clues, a thirty-eighter having already been produced by others.

Developing the Code
After writing the initial non-optimized code, we worked on performance improvements in three steps:

- **Step 1:** Changing the algorithm to shortcut the brute force approach
- **Step 2:** Optimizing the serial code, taking advantage of SSE instructions.
- **Step 3:** Adding parallelism

The code was developed over two years, much of the work done in out-of-office hours, consuming 2,000-3,000 programming hours.

Step 1: Changing the Algorithm

Rather than using a brute force method to create all the different possible solutions, we decided to take an existing puzzle, remove one or two clues, and then use a recursive solver to produce a new puzzle.

To find a new 17-clue puzzle, we start with an 18-clue puzzle, remove two clues, and search for any valid solutions. So for example, in the puzzle below in Figure 1, clues 3 and 9 are first removed from column 1. The solver then populates each of the unsolved cells with a list of valid alternatives.

The solver then recursively prunes down the alternatives to find a valid puzzle, taking care there are no redundant clues. This method of creating a new puzzle we call the "-2 + 1" algorithm.

We use the same technique to find the 'thirty-niner'. Taking an existing 'thirty-eighter, we remove one clue and then add two new clues—we call this a "-1 + 2" algorithm.

Step 2: Optimizing the Serial Code

Modern CPUs have instructions that can work on more than one data item at the same time, that is, Single Instruction Multiple Data (SIMD). Replacing traditional instructions with SIMD instructions can lead to code that runs much faster. Examples of such instructions include MMX and the various Streaming SIMD Extension (SSE, SSE2, ...) .

Adding SSE Intrinsics

SSE intrinsics are compiler-generated, assembler-coded functions that can be called from C\C++ code and provide low-level access to SIMD functionality without the need to use inline assembler. Compared to using inline assembler, intrinsics can improve code readability, assist instruction scheduling, and help reduce debugging effort. Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C and C++ languages.

The Intel® Compiler supports a wide range of architectural extensions from the early MMX instructions to the latest generation of SSE4.2 instructions.

The code in Figure 2 shows how 128-bit SSE2 registers are used in the Sudoku code.

The first version of the Sudoku generator did not use SSE instructions or intrinsics. Reworking the first version of the code to use SSE2 registers took a significant amount of time. The adding of SSE intrinsics gave us a speedup of several hundred.

Using SSE intrinsics does have its drawbacks. You can end up locking your implementation to a particular generation of architecture. The long names of the SSE functions can make your C++ code almost unreadable, and there is a significant learning curve the programmer has to climb. In the case of the Sudoku generator, the performance improvement far outweighed the extra effort that was needed.

Step 3: Adding Parallelism

We used OpenMP* tasks - which are defined in the OpenMP 3.0* standard. OpenMP 3.0 is supported by the Intel® C\C++ Compiler beginning with version 11. Adding the parallelism took about two weeks of work – which felt a lot of effort at the time, but in relation to the length of the project the time was fairly short and well spent. Figure 3 gives an example of using OpenMP tasks.

One of the most difficult aspects of adding the OpenMP code was to grasp how variables were treated. Data in OpenMP can be shared or private. The fine-tuning of our code to get the right scope level for our variables took several iterations. Much of the time taken in adding the parallelism was reworking the code so that there was less need to share data between the different running tasks, and making sure that there were no dependencies between the different loops that were parallelized.

Using Intel® Cilk Rather than OpenMP

Today, Intel® Parallel Studio supports a number of different ways of parallelizing a program. The diagram in Figure 4 shows Intel® Parallel Building Blocks, which offers multiple support for parallel programming.

As stated earlier, the Sudoku thirty-niner was parallelized using OpenMP, it being much easier to use than native threads.

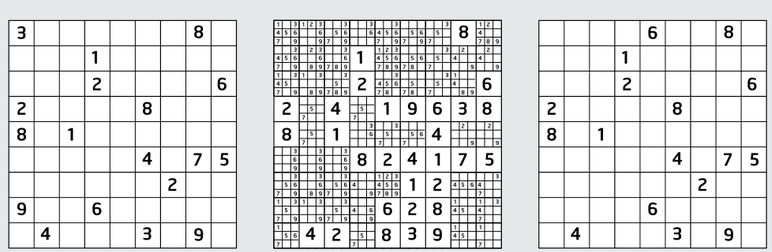


Figure 1: Creating a new 17-clue Sudoku

```
for(int num=0; num < 9; num++)
{                                                    xmm0 becomes a bitmask for all
    __m128i xmm0 = _mm_and_si128(BinSmallNum, BinNum[num]);   occurrences of a particular number
    for(int i=0; i < 9; i++)
    {                                                i is used to traverse each row, column and box
     __m128i BoxSum    =   _mm_and_si128(BinBox[i],    xmm0);
     __m128i RowSum    =   _mm_and_si128(BinRow[i],    xmm0);
     __m128i ColumnSum =   _mm_and_si128(BinColumn[i], xmm0);
     if (ExactlyOneBit(BoxSum))
     {
        int cell=BitToNum(BoxSum);                   Test to see if exactly one
        FoundNumber(cell, num);                      number is used
        return true;
        }
     }
}
```

Figure 2: Using SSE compiler intrinsics to improve performance

If the project was starting again today, then using Intel® Cilk Plus would be an attractive choice. Cilk is one of the easiest ways to parallelize an existing program. Adding Cilk to existing C code is extremely easy.

Cilk uses three key words: cilk_spawn, cilk_sync, and cilk_for. Once the header file cilk.h is included in a file, the key words are available for use. See Figure 5.

In Cilk the programmer does not control the parallelism of a program but rather expresses intent. By placing Cilk keywords in a program, the developer is giving permission for the code to be run in parallel. The decision whether or not to run code in parallel is made by the Cilk scheduler at runtime. The Intel Cilk Plus runtime automatically takes care of load balancing.

Figure 6 shows how the Sudoku code is made parallel using the Cilk cilk_for keyword. The code is inserted in the same place as the original OpenMP tasks (see Figure 3). The solution is embarrassingly simple!

Intel Cilk Plus Is the Easiest Way to Parallelize an Existing Program

When making code parallel, the programmer has to be careful that no data races are introduced. If there are any global variables then the code should be reworked so that the scope of the variables is restricted—by, for example, using local or automatic variables rather than global variables. If it is impossible to remove all global variables, then access to such variables should be protected so that only one thread at a time can modify them.

In Cilk the easiest way to deal with global and shared variables is to declare them to be a reducer. When a worker accesses a reducer it is given its own private view that it can safely manipulate. Views are then later merged in the serial part of the code by a call to get_value(). The code in Figure 7 shows how the global gNumCilkPuzzlesSolved is declared to be a reducer_opadd. The call to get_value(),which is called from the serial part of the code, combines all the values of the reducer, thus obtaining the correct value.

```
#pragma omp parallel                          ← A pool of threads is created here
{
    #pragma omp single nowait                 ← One thread gets to execute the for loop
    {
        for( int i=0; i< NUM_NODES -1; i++)
        {
            NODE Node1 = pPuzzle ->Nodes [i];
            if (Node1.number > 0)
        {
            //create copy of the top level node;
            memcpy (&gPuzzles[i];pPuzzle, sizeof (SUDOKU));
            #pragma omp taskprivate (i)
            GenDoWork (&gPuzzles[i],i;
        }                                     ← The single "for loop" thread creates a task
    }                                           for each instance of GenDoWork ()
}
}
```

Figure 3. Code using OpenMP tasks

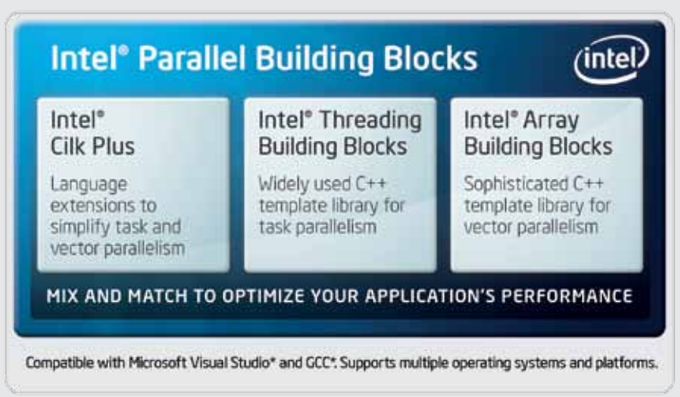


Figure 4. Intel Parallel Building Blocks offers multiple models for parallel programming.


```
#include <cilk/cilk.h>
void work(int num}
{
    // add code here
}

void func1()
{
        cilk_spawn work(1);
        work(2);
        cilk_sync;
}

void func2()
{
    cilk_for(int i=0; i<9; i++)
    {
        work(3);
    }

}
```

cilk_spawn and cilk_sync
The lines between cilk_spawn and cilk_sync are known as the continuation. The cilk_spawn gives permission to the runtime to run work(1) in parallel with the continuation code. If there is a spare worker available, the scheduler steals the continuation code from the first worker and assigns it to a second worker—at the same time the first worker continues executing work(1). After cilk_sync the code reverts to serial execution

cilk_for
Replaces a standard C/C++ for loop. The loops are shared between available workers. No particular order of execution is guaranteed. Once all the loops have been executed the program continues. If loops do unequal amounts of work, load balancing will be taken care of by the scheduler's work-stealing algorithm.

Figure 5. The three Intel Cilk Plus keywords

```
#include <cilk/cilk.h>
.
.
.
cilk_for(int i = 0 ; i < NUM_NODES -1; i++ )

    {
        NODE Node1 = pPuzzle->Nodes[i];
        if(Node1.number > 0)
        {
            // create a copy of the top level node;
            memcpy(&gPuzzles[i],pPuzzle,sizeof(SUDOKU));
            GenDoWork(&gPuzzles[i],i);
        }
     }
```

Figure 6. Adding cilk_for to the Sudoku code

"Whether optimizing serial code or creating new applications for multicore, parallel programming has evolved to improve proficiency with flexible design models and tools that make applications reliable and fast."

```
int gNumCilkPuzzlesSolved;      // global variable
.
.
gNumCilkPuzzlesSolved++;         // somewhere in parallel code
.
.
int Tmp = gNumCilkPuzzlesSolved; // somewhere in serial code

(a) The global variable gNumCilkPuzzlesSolved is unsafe to use in parallel code.

#include <cilk/reducer_oppad.h>
cilk::reducer_opadd<int> gNumCilkPuzzlesSolved;
.
.
gNumCilkPuzzlesSolved++;          // in parallel code
.
.
int Tmp = gNumCilkPuzzlesSolved.get_value();// in serial code


(b) The  global variable is declared to be a reducer, making it safe to access.
```

Figure 7. Fixing data race problems by using Intel Cilk Plus reducers

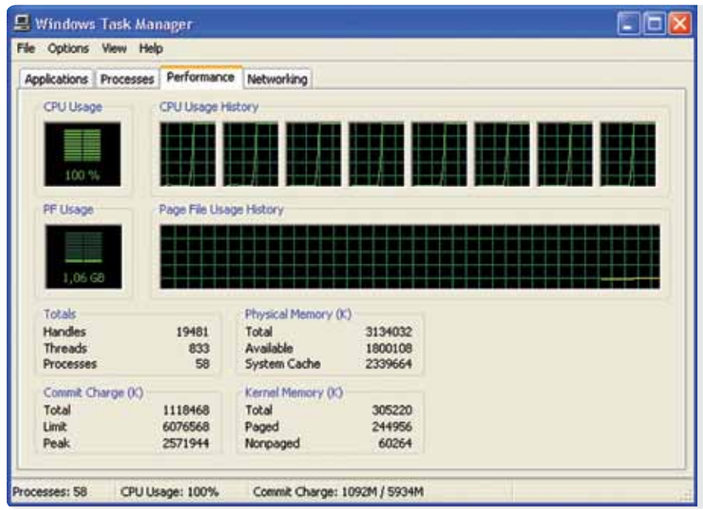
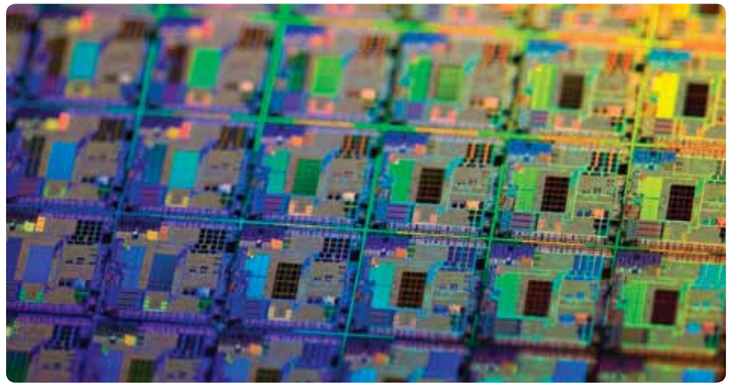


Figure 8. With the parallelism implemented in the code, each hardware thread ran at 100 percent use.



The Results

In our original OpenMP solution once the parallel code was added to the project, it was rewarding to see that on an SMT quad core—which can support eight hardware threads—all eight hardware threads were kept busy. See Figure 8.

Later experimentations using Cilk showed that the Cilk solution performed equally well to the OpenMP one, and was much easier to implement.

Figure 9 shows the three new "thirty-niners" that were found using our Sudoku generator. □

				3				
		3	6		7			1
6		4		9	1	3		7
5					3		2	4
7	4			6	2	5		3
		2			5	7	1	
2		5	7	1	6	4		
4		6		2	9	1	7	5

				3				
		3	6		7			1
6		4		9	1	3		7
5					3		2	4
1	4			6	2	5		3
		2			5	1	7	
2		5	1	7	6	4		
4		6		2	9	7	1	5

		1		3	7	6	4	
							8	
6		4		9	1	3		7
		2			6			3
1	4			2	3	5		6
3				1	5	4	2	
2		5			9	7		4
4	1			7	2			
	7			5		2	6	

Figure 9. The three 39 minimal solutions found using the minus-one-plus-two search

Intel® Cilk Plus

Intel® Cilk Plus is made up of these main features:

- A set of keywords, for expression of task parallelism
- Reducers, which eliminate contention for shared variables among tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion
- Array notations, which provide data parallelism for sections of C\C++ notation to manipulate arrays
- Elemental functions, which enable data parallelism of whole functions or operations which can then be applied to whole or parts of arrays or scalars
- The simd pragma, which lets you express vector parallelism for utilizing hardware SIMD parallelism while writing standard compliant C/C++ code with an Intel® Compiler.

** Lars Peters Endresen and Håvard Graff wrote the original program using OpenMP. The design was later rewritten by Stephen Blair-Chappell using Intel® Cilk Plus.

The complete case study, along with hands-on examples, will be available in the WROX book *Parallel Programming with Intel Parallel Studio*. Stephen Blair-Chappell and Andrew Stokes, Wiley Publishing Inc. ISBN 9780470891650 (March 2011)

WHICH COMES FIRST:
parallel languages or parallel programming patterns?

CLAY BRESHEARS,
COURSEWARE ARCHITECT/INSTRUCTOR,
INTEL CORPORATION

On the shuttle to the recent UPCRC (Universal Parallel Computation Research Center) Annual Summit meeting on the Microsoft* campus in Redmond, WA, I was listening in on a discussion about parallel programming patterns. Being a parallel programmer, I was interested in what people (and these were some of the experts in the field) had to say about parallel programming patterns, how they are evolving, and how they will impact future parallel coders. The discussion turned to whether patterns would affect programming languages directly or remain something that would be constructed from statements of the language. I think I'm in the former camp. **Here's why.**

For those of us that were programming when Elvis was still alive, think back to writing with assembly language. For the most part, there were instructions for Load, Store, Add, Compare, Jump, plus some variations on these and other miscellaneous instructions. To implement a counting/indexing loop you would use something like the following:

```
Initialize counter
LOOP: test end condition,
       goto EXIT if done
Loop Body
increment counter
goto LOOP
EXIT: next statement
```

This is a programming pattern. (Surprised?) With the proper conditional testing and jumping (goto) instructions within the programming language, this pattern can be implemented in any imperative language.

Since this pattern proved to be so useful and pervasive in the computations being written, programming language designers added syntax to "automate" the steps above. For example, the for-loop in C.

```
for (i = 0; i < N; ++i) {
Loop Body
}
```

Once we had threads and the supporting libraries to create and manage threads, parallel coding in shared memory was feasible, but at a pretty crude level since the programmer had to be sure the code handled everything explicitly. For example, dividing the loop iterations among threads can be done with each thread executing code that looks something like this:

```
start = (N/num_threads) * (myid)end =
(N/num_threads) * (myid + 1)
if (myid == LAST) end = N
for (i = start; i < end; ++i) {
Loop Body
}
```

Parallel programming patterns will be abstractions that can be "crudely" implemented in current languages and parallel libraries, like the pseudocode above. New languages (or language extensions) will make programming parallel patterns easier and less error prone. From the example above, OpenMP* has the syntax to do this, but it only takes a single line added to the serial code:

```
#pragma omp for
for (i = 0; i < N; ++i) {
Loop Body
}
```

From the evidence above, I think future parallel programming languages or language extensions supporting parallelism will be influenced by the parallel programming patterns we define and use today. And nothing will remain static. Ralph Johnson (Design Patterns), during his presentation, remarked that some of the original patterns saw early use, but this use has slacked off. Two reasons he noted for this was that some of the patterns couldn't easily be implemented in Java* and modern OO languages had better ways to accomplish the same tasks—most likely these new languages found inspiration from the patterns and their usage.

For an answer to the question posed in the title, it boils down (no pun intended) to the old chicken-and-egg paradox. There were algorithms (patterns) to do computations before there were computers; prior to that, those algorithms were modifications of previous algorithms influenced by the tools available. Looking forward, though, we're still in the relative stages of infancy for programming, let alone parallel programming. Clearly, the next generation of parallel programming languages or libraries or extensions bolted onto serial languages will be influenced by the patterns we use now for specifying parallel computations.

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

For more information regarding performance and optimization choices in Intel software products, visit http://software.intel.com/en-us/articles/optimization-notice.

The Ultimate All-in-One Performance Toolkit

Introducing Intel® Parallel Studio 2011

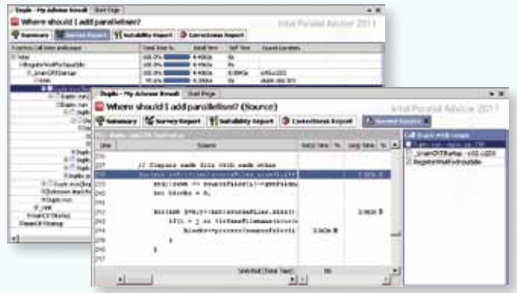
Intel® Parallel Studio 2011 simplifies and speeds the analysis, compiling, debugging, error-checking, and tuning of your serial and parallel apps. With Intel Parallel Studio, you get everything you need to optimize legacy serial code, exploit multicore, and scale for manycore.

INTEL® PARALLEL STUDIO COMPONENTS

Intel Parallel Studio supports every stage of the development lifecycle.

Innovative threading assistant

Intel® Parallel Advisor 2011: Follow the steps for task-oriented parallelism and target coarse-grain parallelism.

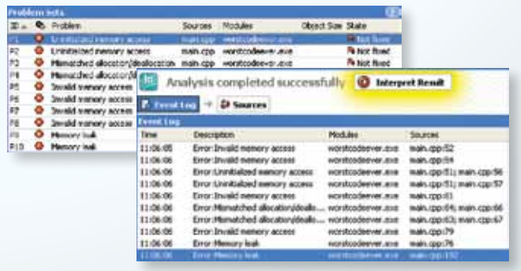


Optimizing compiler and threaded libraries

Intel® Parallel Composer 2011: A simple recompile with Intel Parallel Composer can yield better performance.

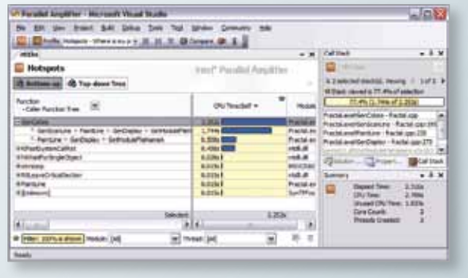
Memory and threading error checker

Intel® Parallel Inspector 2011: Quickly find memory errors in your single and multithreaded applications.



Threading and performance profiler

Intel® Parallel Amplifier 2011: Find the functions in your application that consume the most time.



Rock your code.

ROCK YOUR WORLD.

If you own the original Intel Parallel Studio, upgrade for free. Learn more and check out what's new in Intel Parallel Studio 2011 at http://software.intel.com/en-us/intel-parallel-studio-home/.